CS312

OpenGL

Viewing Transformations and Projections

Controlling states

- Enabling features

```
glEnable(GL_DEPTH_TEST);
```

- Setting state

```
glShadeModel(GL_FLAT);
glShadeModel(GL_SMOOTH);
```

OpenGL Buffers

- Color buffer
  - Front and back
- Depth buffer (z-buffer)
  - Hidden surface removal
- Clearing buffers
  - `glClearColor(r,g,b,a);`
  - `glClearDepth(1.0);`
  - `glClear(GL_COLOR_BUFFER_BIT| GL_DEPTH_BIT);`

Depth Buffering

- Request a depth buffer

```
glutInitDisplayMode(GLUT_DEPTH|...);
```

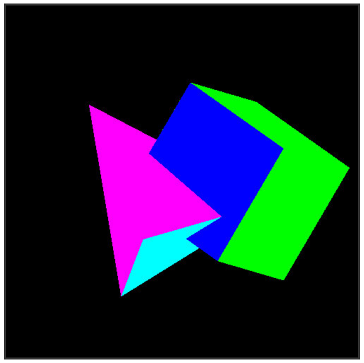
- Enable depth buffering

```
glEnable(GL_DEPTH_TEST);
```

- Clear color and depth buffers

```
glClear(GL_COLOR_BUFFER_BIT |
   GL_DEPTH_BUFFER_BIT);
```

- Render scene
- Swap color buffers

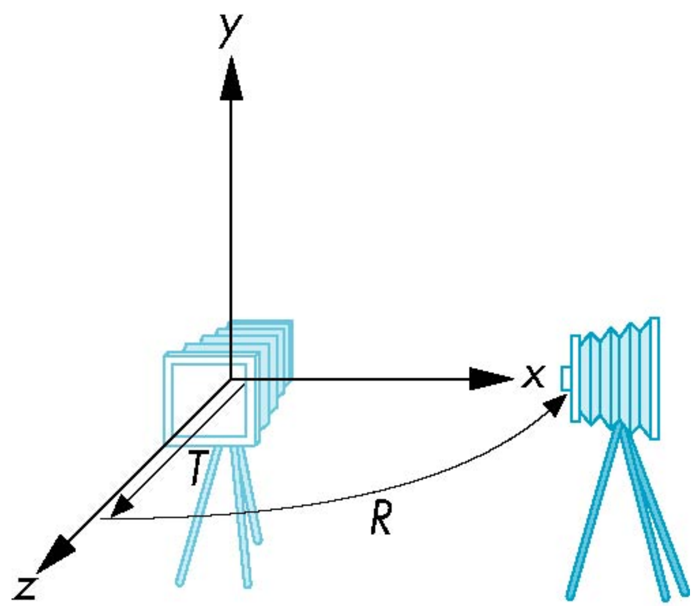


Moving the Camera

- The First Approach:
  - Specify the position indirectly by applying a sequence of rotations and translations to the model-view matrix.
  - This is a direct application of the geometric transformations.

Moving the Camera

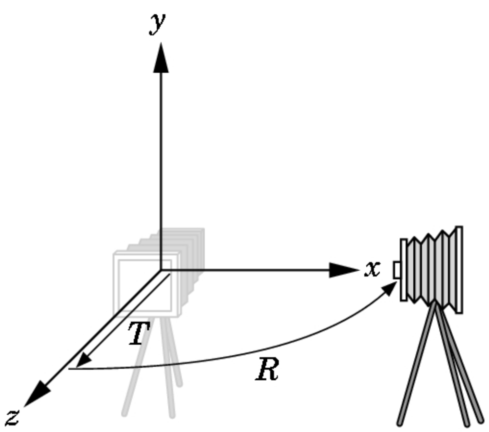
- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view
  - Rotate the camera
  - Move it away from origin
  - Model-view matrix C = TR



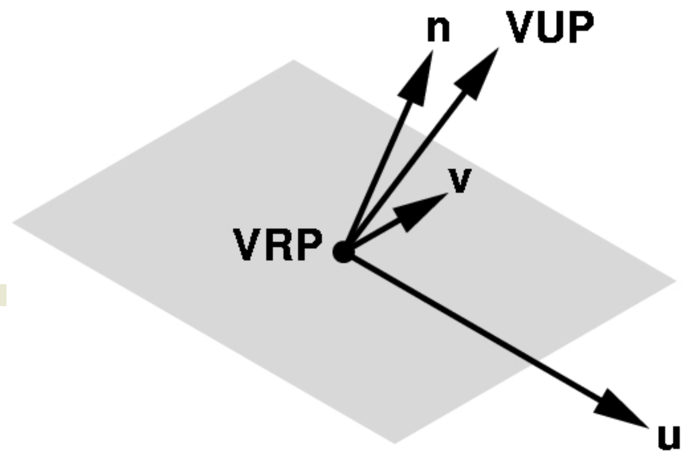
Moving the Camera

- We must be careful for two reasons:
  - First, we usually want to define the camera before we position the objects in the scene.
  - Second, transformations on the camera may appear to be backward from what we might expect.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity( );
glTranslatef(0.0, 0.0, -d);
glRotatef(-90.0, 0.0, 1.0, 0.0)
```



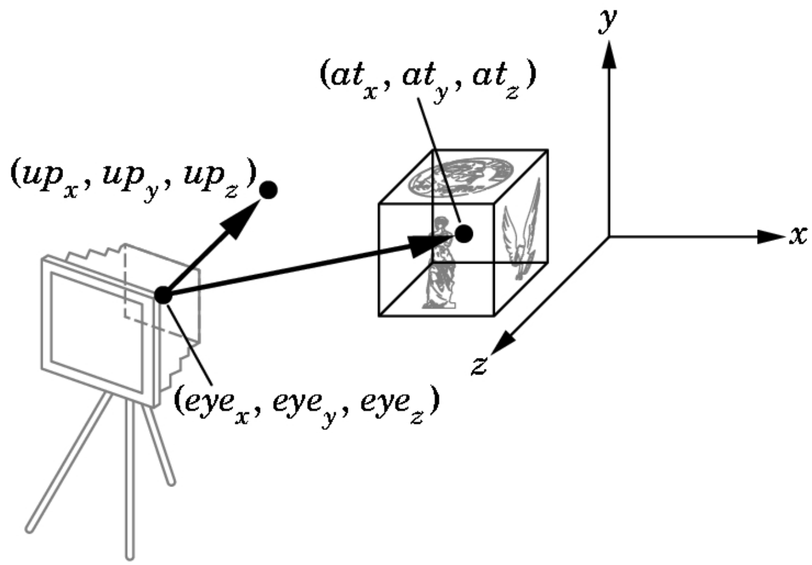
Viewing APIs



- We can take a different approach to positioning the camera – We describe the camera's position and orientation in the world frame
  - It's desired location is centered at the view-reference point (VRP)
  - It's orientation is specified with the view-plane normal (VPN) and the view-up vector (VUP)

gluLookAt

- GL uses a more direct method, fortunately.



- **`gluLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz);`**

gluLookAt

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity( );
gluLookAt(…);

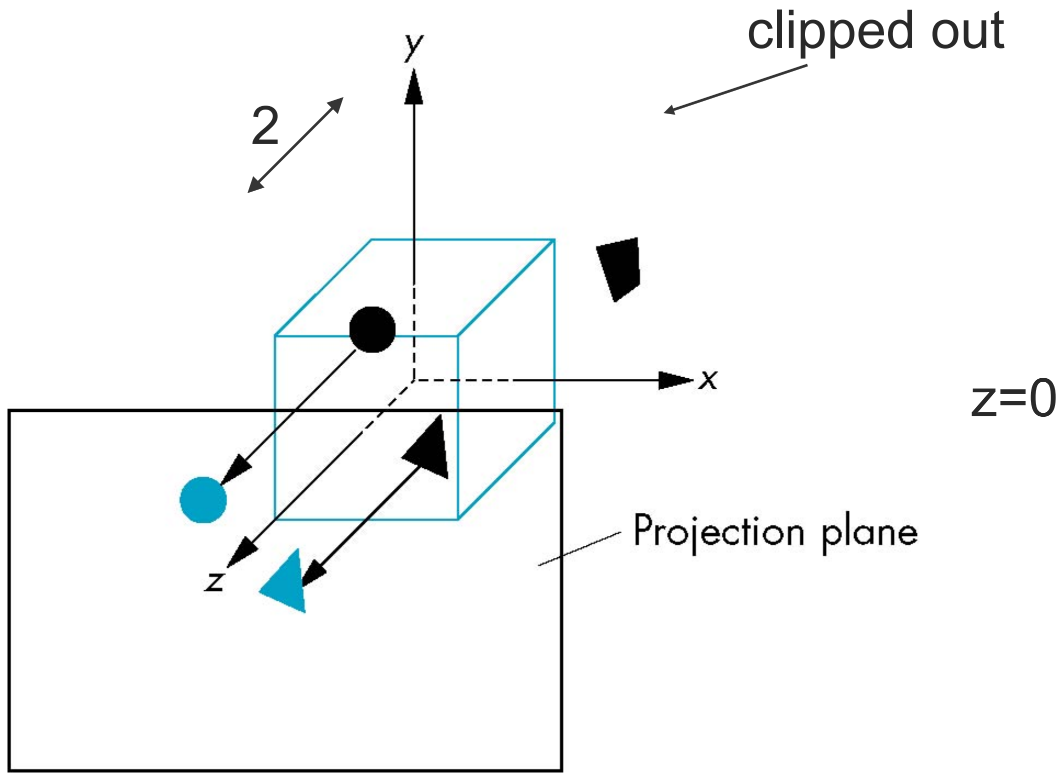
//transformations
//draw ojects
```

The OpenGL Camera

- In OpenGL, initially the world and camera frames are the same
  - Default model-view matrix is an identity
- The camera is located at origin and points in the negative z direction
- OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity

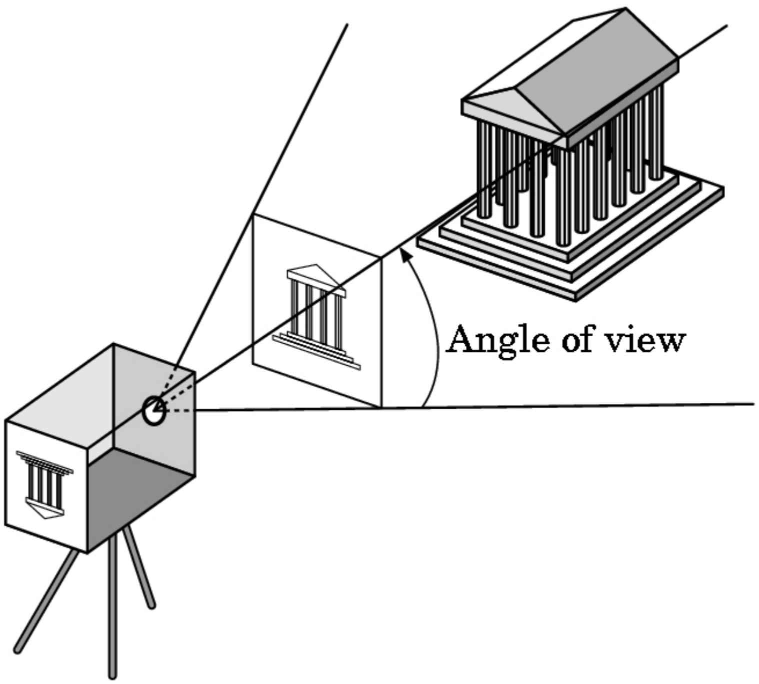
Default Projection

Default projection is orthogonal



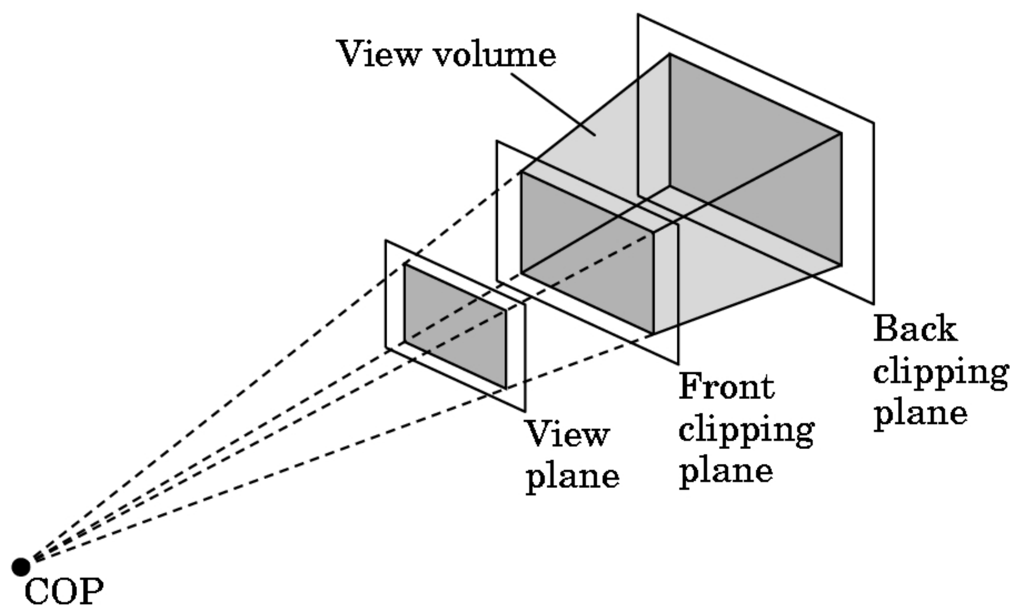
Projections in OpenGL

- The View Volume



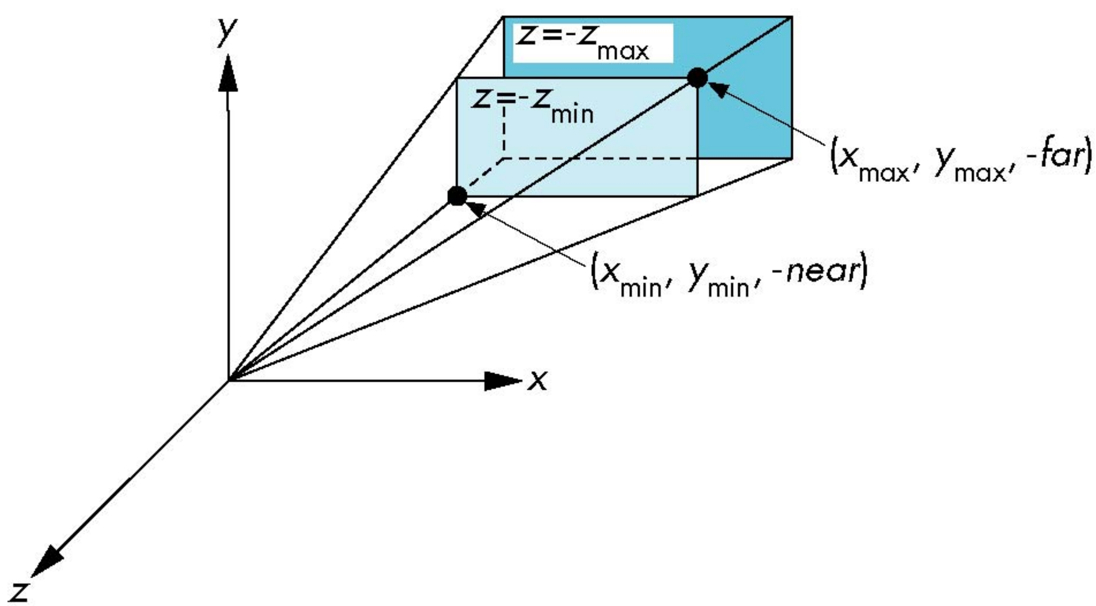
Frustum

- Define clipping parameters through the specification of a projection.
- The resulting view volume is a frustum – which is a truncated pyramid.



Perspectives in OpenGL

- OpenGL has two functions for specifying perspective views
  - **`glFrustum(xmin, xmax, ymin, ymax, near, far);`**



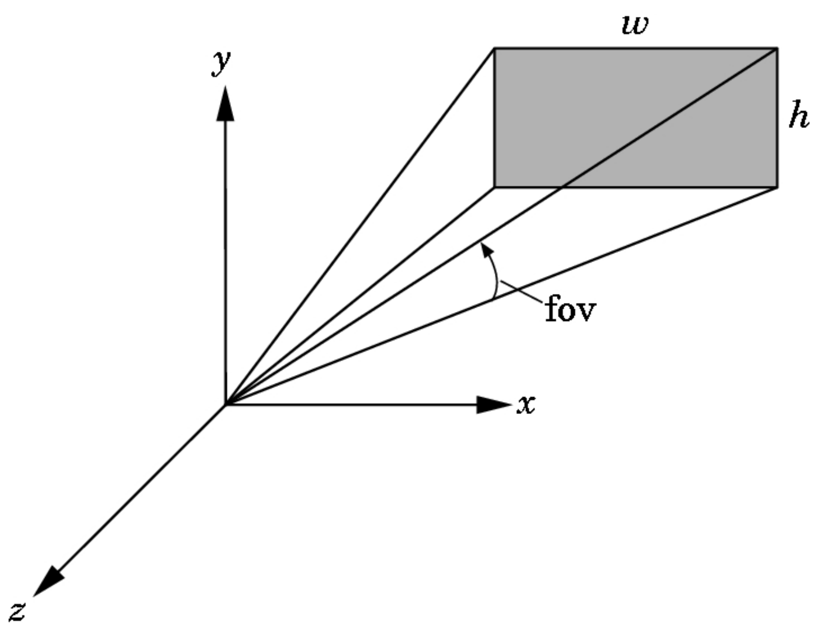
Current Matrix

- The projection matrix determined by these specifications multiplies the present matrix.
- A typical sequence

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(xmin, xmax, ymin,
   ymax, near, far);
```

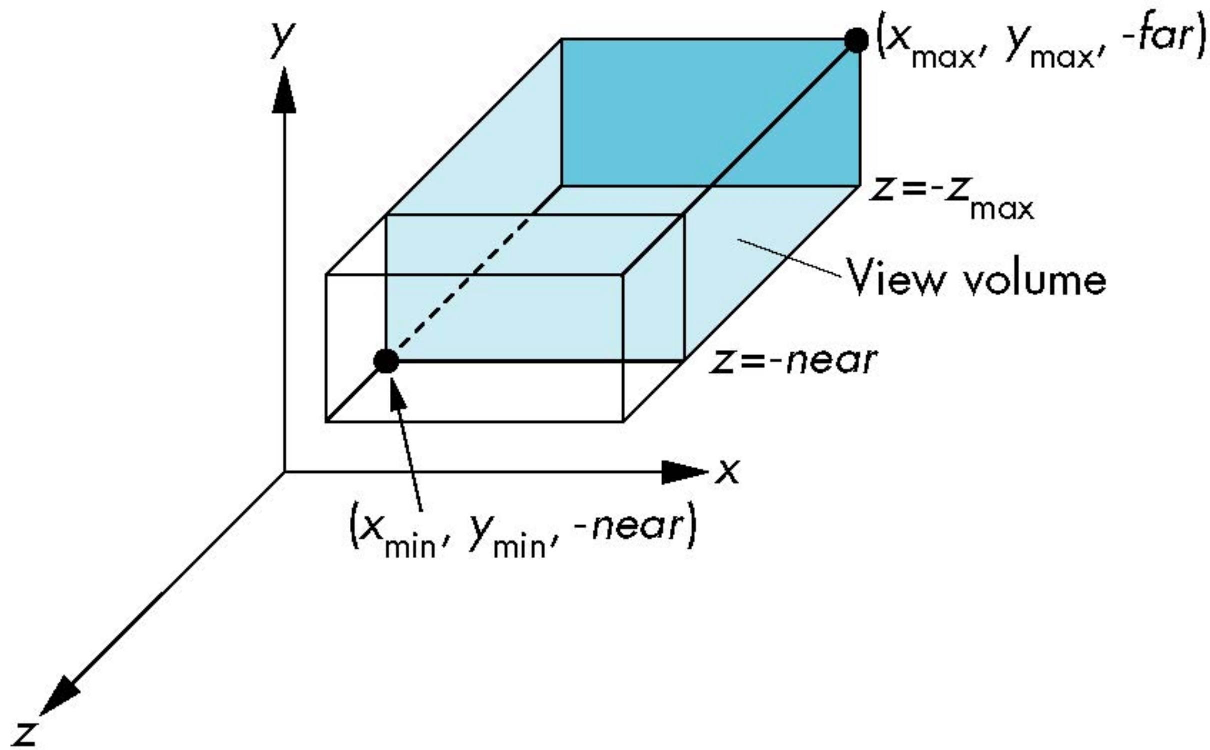
Field of View

- ```
  gluPerspective(fovy, aspect,
  near, far);
  ```


  ```

Parallel Viewing in OpenGL

- `glOrtho(xmin, xmax, ymin, ymax, near, far);`



glut 3D Primitives

- Cube
  - `void glutSolidCube(GLdouble size);`
  - `void glutWireCube(GLdouble size);`
- Sphere
  - `void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);`
  - `void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);`

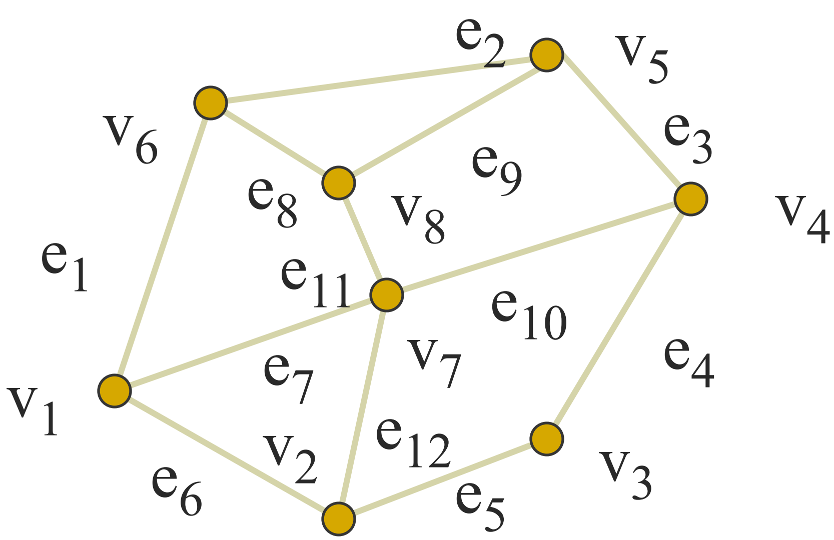
glut 3D Primitives

- Teapot
  - `void glutSolidTeapot(GLdouble size);`
  - `void glutWireTeapot(GLdouble size);`
- Many other geometric shapes

Defining your own shapes

- Objects are surfaces – hollow inside
- Objects are approximated by flat, convex polygons
- Each of these polygons (faces) is given by a set of 3D vertices
- This set of vertices and how they connect (edges) is known as a mesh

Representing a Mesh



- There are 8 nodes and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges
- Each vertex has a location $v_i = (x_i\ y_i\ z_i)$

Simple Representation

- Define each polygon by the geometric locations of its vertices

```
glBegin(GL_POLYGON);
    glVertex3f(x1, y1, z1);
    glVertex3f(x2, y2, z2);
    glVertex3f(x7, y7, z7);
glEnd();
```

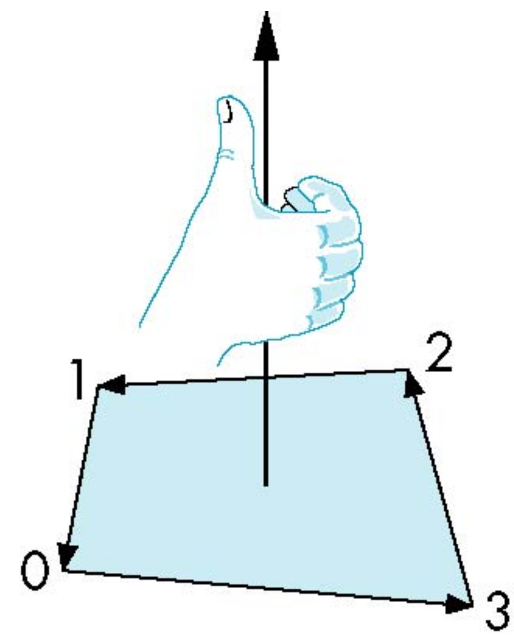
- Inefficient and unstructured
  - Consider moving a vertex to a new location

Inward and Outward Facing Polygons

- $\{v_0, v_3, v_2, v_1\}$ and $\{v_1, v_0, v_{3,} v_2\}$ are equivalent in that the same polygon will be rendered by OpenGL but the order $\{v_0, v_1, v_{2,} v_3\}$ is different
- The first two describe *outwardly facing* polygons
- OpenGL can treat inward and

outward facing polygons differently

- Use the *right-hand rule* =>



Geometry vs Topology

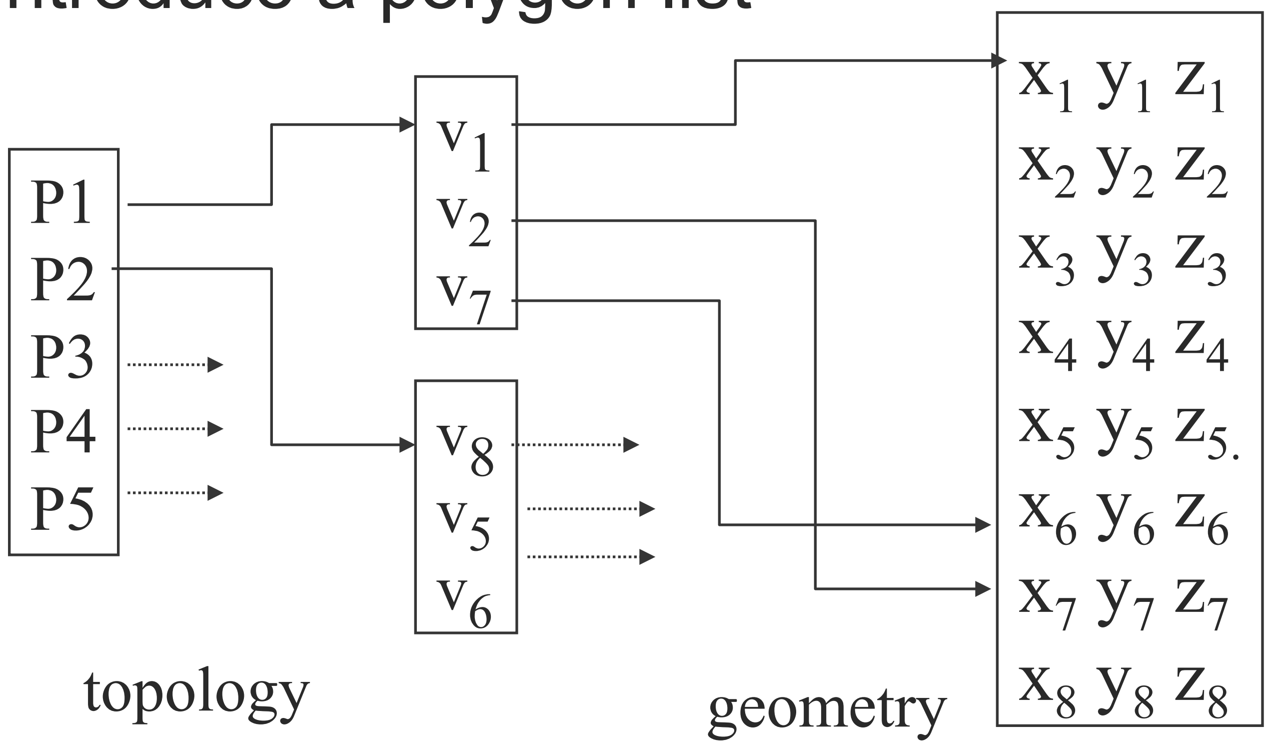
- Generally it is a good idea to look for data structures that separate the geometry from the topology
  - Geometry: locations of the vertices
  - Topology: organization of the vertices and edges
  - Topology holds even if geometry changes

Geometry vs Topology

- Example: a cube can be specified with **`GL_QUADS`** or **`GL_POLYGON`** 6 times
- Fails to capture the topology
  - A polyhedron with 6 faces.
  - Each face has 4 vertices
  - Each vertex share 3 faces

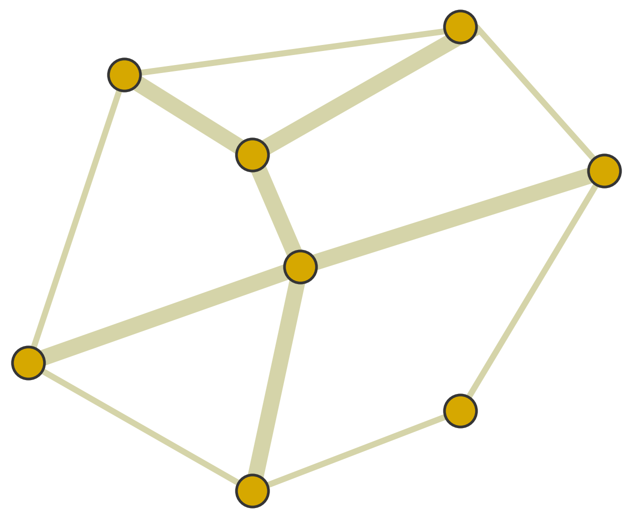
Vertex Lists

- Put the geometry in an array
- Use pointers from the vertices into this array
- Introduce a polygon list



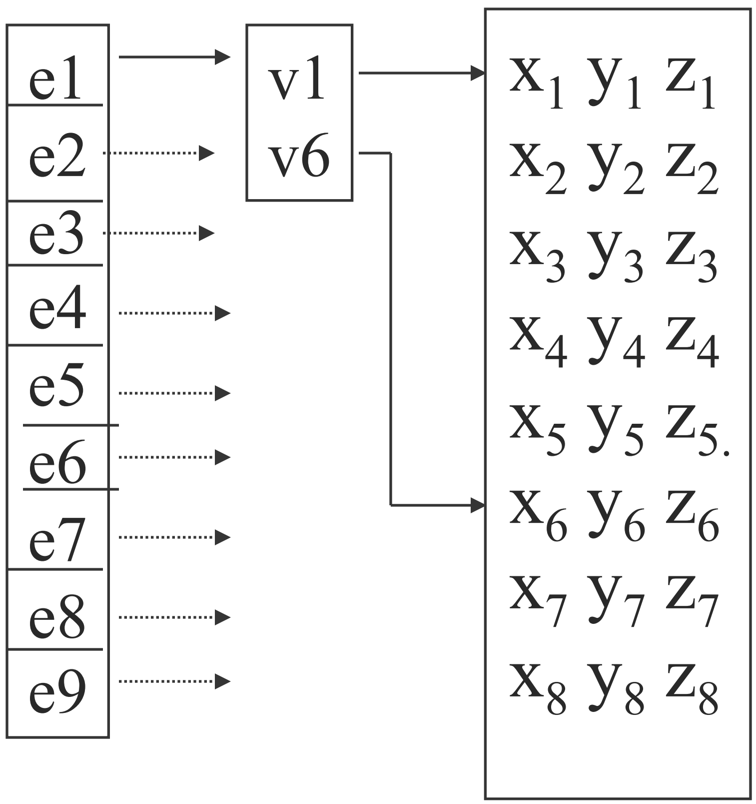
Shared Edges

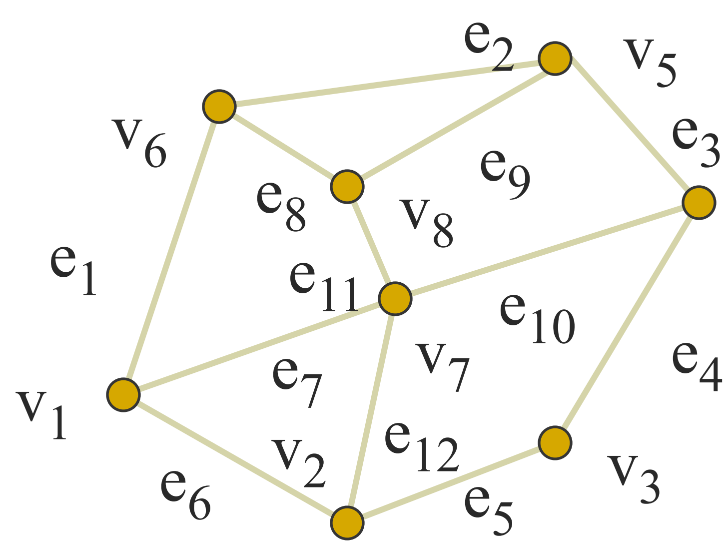
- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice



- Can store mesh by *edge list*

Edge List





Note polygons are not represented

Modeling a Cube

```
GLfloat vertices[][3] =
{{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
 {1.0,1.0,-1.0},{-1.0,1.0,-1.0},{-1.0,-1.0,1.0},
 {1.0,-1.0,1.0},{1.0,1.0,1.0},{-1.0,1.0,1.0}};

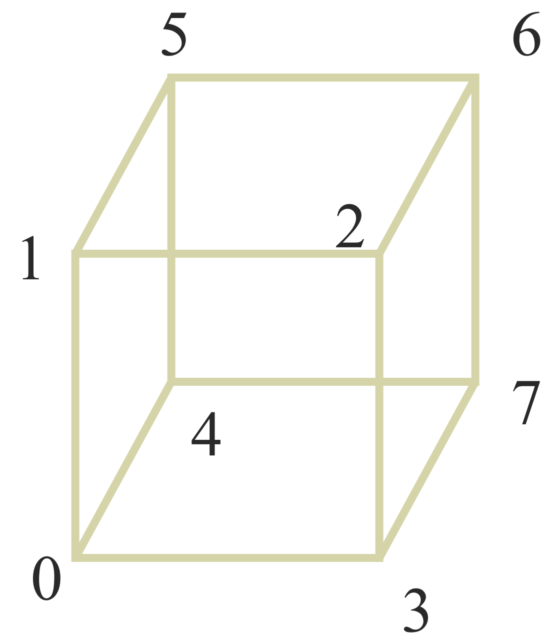
GLfloat colors[][3] =
{{0.0,0.0,0.0},{1.0,0.0,0.0},
 {1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
 {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

Drawing a polygon from a list of indices

```
void polygon(int a, int b, int c , int d){
   glBegin(GL_POLYGON);
      glColor3fv(colors[a]);
      glVertex3fv(vertices[a]);
      glVertex3fv(vertices[b]);
      glVertex3fv(vertices[c]);
      glVertex3fv(vertices[d]);
    glEnd();
  }
```

Draw cube from faces

```
void colorcube(){
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
```



Note that vertices are ordered so that
we obtain correct outward facing normals

Efficiency

- The weakness of our approach is that we are building the model in the application and must do many function calls to draw the cube
- Drawing a cube by its faces in the most straight forward way requires
  - 6 **`glBegin`**, 6 **`glEnd`**
  - 6 **`glColor`**
  - 24 **`glVertex`**
  - More if we use texture and lighting

Vertex Arrays

- OpenGL provides a facility called *vertex arrays* that allows us to store array data in the implementation
- Six types of arrays supported
  - Vertices
  - Colors
  - Color indices
  - Normals
  - Texture coordinates
  - Edge flags
- We will need only colors and vertices

Initialization

- Using the same color and vertex data, first we enable

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

- Identify location of arrays

```
glVertexPointer(3, GL_FLOAT, 0, vertices);
```

3d arrays | stored as floats | data contiguous | data array

```
glColorPointer(3, GL_FLOAT, 0, colors);
```

Mapping indices to faces

- Form an array of face indices

```
GLubyte cubeIndices[24] = {0,3,2,1,
                           2,3,7,6
                           0,4,7,3,
                           1,2,6,5,
                           4,5,6,7,
                           0,1,5,4};
```

- Draw through **`glDrawElements`** which replaces all **`glVertex`** and **`glColor`** calls in the display callback

Drawing the cube

- Method 1:

what to draw — number of indices

```
for(i=0; i<6; i++)
  glDrawElements(GL_POLYGON, 4,
      GL_UNSIGNED_BYTE, &cubeIndices[4*i]);
```

format of index data — start of index data

- Method 2:

```
glDrawElements(GL_QUADS, 24,
    GL_UNSIGNED_BYTE, cubeIndices);
```

Draws cube with 1 function call!!

Idle Callback

- Minimize the amount of computation done in an idle callback.
- If using idle for animation, stop rendering when nothing changed, or window not visible

```
glutVisibilityFunc(visible);
void visible(int vis) {
  if (vis == GLUT_VISIBLE)
   glutIdleFunc(idle);
  else
   glutIdleFunc(NULL);
}
```

Back Face Culling

- OpenGL can compute and remove those faces that are facing away from the viewer.
- **`glEnable(GL_CULL);`**

Timer Callback

- **`void glutTimerFunc(unsigned int msecs, void (*func)(int value), value);`**
- Registers the timer callback **`func`** to be triggered in at least **`msecs`** milliseconds.

```
#define FR 60
glutTimerFunc(100, myTimer, 0);
void myTimer(int v) {
  //update and advance states
  glutPostRedisplay();
  glutTimerFunc(1000/FR, myTimer, v);
}
```